ECE 320
Core File Documentation

Daniel Grobe Sachs - January 21, 2001

1 Introduction

In this document, we will introduce the six-channel surround sound board for the TI EVM320C549 and the "core" code that we have written for it. This document details the core code, memory maps, serial port interface and memory interface offered by the core file.

2 The Six-channel Surround Board

The six-channel board attaches to the top of the DSP evaluation module its onboard, one-channel A/D and D/A with 6 channels of D/A and 2 channels of A/D running at a sample rate of 44.1 KHz. Alternatively, the A/D can be disabled and a SP/DIF digital input enabled, allowing PCM digital data from a CD or DVD to be sent directly into the DSP for processing. The two input channels and six output channels all sample at the same time (clock skew between channels is not an issue). Our core code reads and writes blocks of up to 80 samples (for each channel of input or output) at a time; the unmodified core code reads and writes blocks of 64 samples at a time. If a larger aggregation is required, you will need to implement it yourself.[1]

Other features include buffered serial communication code, which allows you to send and receive data from the serial port. This can be used to quickly and easily generate GUI controls for your code; it can also be used to report back status to the PC for applications such as speech recognition.

The core code also initializes the DSP itself. It enables the fractional arithmetic mode for the ALU, programs the wait states for the external memory, and sets the DSP clock to 80MHz. (The DSP is rated to run at 100MHz; however, the serial port does not work reliably when the DSP clock speed is greater than 80MHz.)

2.1 Assembling the 6-Channel Sample Code

We'll start with a sample application, a version of the "thru" code that uses the 6-channel board and the new core. First, copy the `thru6.asm` file from the `v:\ece320\54x\dsplib` directory to a directory on your `W:` drive, and assemble the code:

- ☐ `copy v:\ece320\54x\dsplib\thru6.asm thru6.asm`
- ☐ `asm thru6`

[1] The limit on the block length comes from the 2048 words of memory that can be accessed by the auto-buffering unit. This memory is divided up into 1024-word transmit and receive sections; there's room for 160 6-channel samples in the transmit buffer. Since the buffer must be divided into two parts for double-buffering, the maximum block size is 80 samples.

2.2 Testing the 6-Channel Sample Code

Once you've done that, start Code Composer. Since we're using the on-chip RAM on the DSP320C549 to hold program code, we need to map that RAM into program space before we can load our code. This can be done by opening the CPU Registers window (the same one you use to look at the `ARx` registers and the accumulators) and changing the `PMST` register to `0xFFE0`. This sets the `OVLY` bit to 1, switching the internal RAM into the DSP's program memory space.

Finally, load the `thru6.out` file, use Code Composer's `Reset DSP` menu option to reset the DSP, and run the code. Probe at the connections with the function generator and the oscilloscope; inputs and outputs are shown below. Note that outputs 1-3 come from input 1, and outputs 4-6 come from input 2. Figure 1 shows the 6-channel board's connector configuration.

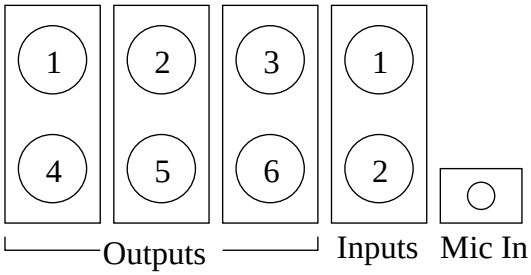

Figure 1: 6-Channel Board Analog Inputs and Outputs

A serial communication program called `Tera Term Pro` has been placed on the V: drive in the directory `V:\ece320\TTERMPRO`.[2] This can be used for testing the serial communication portion of the `thru6` code.

First, ensure that the serial port on the DSP evaluation board has been attached to the computer. The serial port is the 9-pin D-shell connector on the front of the DSP evaluation board (next to the power connector); there should be matching a 9-pin cable already attached to the COM2 port on the back of the computer hanging near the DSP board, if the serial cable is not already attached to the DSP.

Next, start `TTSSH.EXE`. (To do this, simply type `v:\ece320\ttermpro\ttssh` from an MS-DOS command prompt.) When you start TTSSH, it will open a terminal window. This window has been set to communicate with the DSP board on COM2; with the `thru6.asm` code running, the DSP will echo the characters that you type back into this window. (Due to a terminal emulation quirk, the "Enter" key doesn't work properly.)

After you've verified that the EVM is communicating with the PC, close the terminal window.

3 Memory Maps and the Linker

Because the DSP has separate program and data spaces, you would expect for the program and data memory to be independent. However, for the DSP to run at its maximum efficiency, it needs to read its code from on-chip RAM instead of going off-chip, which requires a one- or two-cycle delay whenever an instruction is read. The 32K

[2] Actually, Tera Term Pro is a nice SSH and Telnet client as well. You can Choose "New Connection" under "File", then click on TCP/IP to use it this way. Highly recommended as an alternative to Windows Telnet.

words of on-chip RAM, however, are a single memory block; we cannot map one part of it into program space and another part of it into data space. We can, however, configure the DSP so that the on-chip RAM is mapped into both program space and data space, allowing code to be executed from the onboard memory and avoiding the extra delay. Figure 2 shows the DSP's memory map with the DSP's on-chip memory mapped into program space.

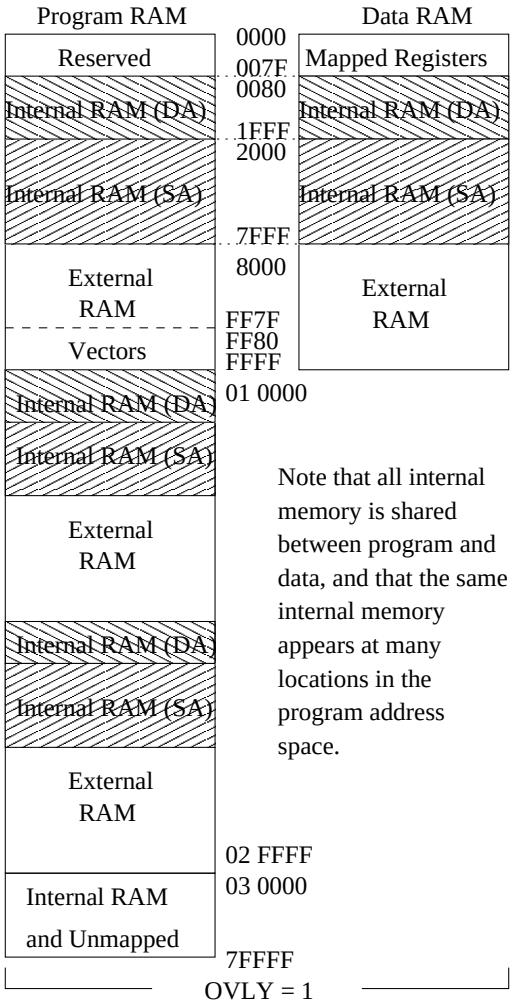

Figure 2: Hardware Memory Map

Because the same on-chip RAM is mapped into both program and data space, care must be taken to not overwrite your code with data or vice versa. To help prevent this, the linker will place the code and data in different parts of the memory map. If you use the `.word` or `.space` directives to inform the linker of all of your usage of data memory, you will not accidently try to reuse memory that has been used to store code or other data. Avoid using syntaxes like `stm #2000h,AR3` to point address registers to specific memory locations directly, as you may accidently overwrite important code or data. Instead, use labels that point to memory spaces explicitly declared by `.word` or `.space` directives in your code.

There are two types of internal memory on the TI TMS320C549 DSP: SARAM (Single Access RAM) and DARAM (Dual Access RAM). The first 8K of internal memory is DARAM; the next 24K is SARAM. The difference between these two types of memory is that while SARAM can only be read or written once in a cycle, DARAM can be read or written twice in a cycle. This is relevant because the TMS320C549 DSP core can access memory up to three times in each cycle - two accesses in Data RAM to read or write operands, and one access in Program

RAM to fetch the next instruction. If, for instance, two operands are fetched from the same "page[3]" of SARAM in the same cycle[4], a one-cycle stall will occur.

Part of the SARAM (from 6000h to 7FFFh) is used for storing your program code; a small amount of SARAM below 6000h is also used for storing the DSP's stack. Part of the DARAM (from 0800h to 0FFFh) is used for the input and output buffers and is also unavailable. Ensure that any code you write does not use any of these reserved sections of data memory. In addition, the core file reserves six locations in scratch-pad RAM (060h to 065h); do not use these in your program code.

4 Sections and the Linker

The recommended way of preventing the memory conflicts outlined above from affecting your code is to ensure that all memory your program uses is allocated using the assembler's `.word` or `.space` directives. (Remember that `.word` allocates one memory location and initializes it to the value given as its parameter. `.space 16*<words>` allocates `<words>` words of uninitialized storage.)

Like before, you can use the `.text` directive to declare program code, and the `.data` directive to declare data. However, there are many more sections defined by the linker control file. Note that the core file uses memory in some of these sections.

You can place program code in the following sections using the `.sect "<section>"` directive:

- .text: (`.sect ".text"`) SARAM between 6000h and 7FFFh (8192 words)
- .etext: (`.sect ".etext"`) External RAM between 8000h and FEFFh (32,512 words). The test-vector version of the DSP core stores the test vectors in the `.etext` section.

You can place data in the following sections:

- .data: (`.sect ".data"`) DARAM between 1000h and 1FFFh (4096 words)
- .sdata: (`.sect ".sdata"`) SARAM between 2000h and 5EFFh (24,320 words)
- .ldata: (`.sect ".ldata"`) DARAM between 0080h and 07FFh (1,920 words)
- .scratch: (`.sect ".scratch"`) Scratchpad RAM between 0060h and 007Fh (32 words)
- .edata: (`.sect ".edata"`) External RAM between 8000h and FFFFh (32,768 words) [5]

If you always use these sections to allocate data storage regions instead of setting pointers to arbitrary locations in memory, you will not accidently overwrite your program code or important data stored at other locations in

[3] SARAM is divided into 8K word pages: 2000h-3FFFh, 4000h-5FFFFh, and 6000h-7FFFh. Two locations in different pages can be accessed simultaneously without conflict. DARAM is also split into pages (2K words each); this allows the auto-buffering unit to access its data without conflicting with reads or writes occurring elsewhere in DARAM.

[4] Due to the pipeline, two memory accesses in the same instruction execute in different cycles. However, if two successive instructions access the same area of SARAM, a stall can occur.

[5] May not currently be working - we're looking into it.

memory. However, the linker cannot prevent your pointers from being incremented past the end of the memory areas you have allocated.

Figure 3 shows the memory regions and sections defined by the linker control file. Note that the sections defined in the linker control file but not listed above are reserved by the core file and should not be used.

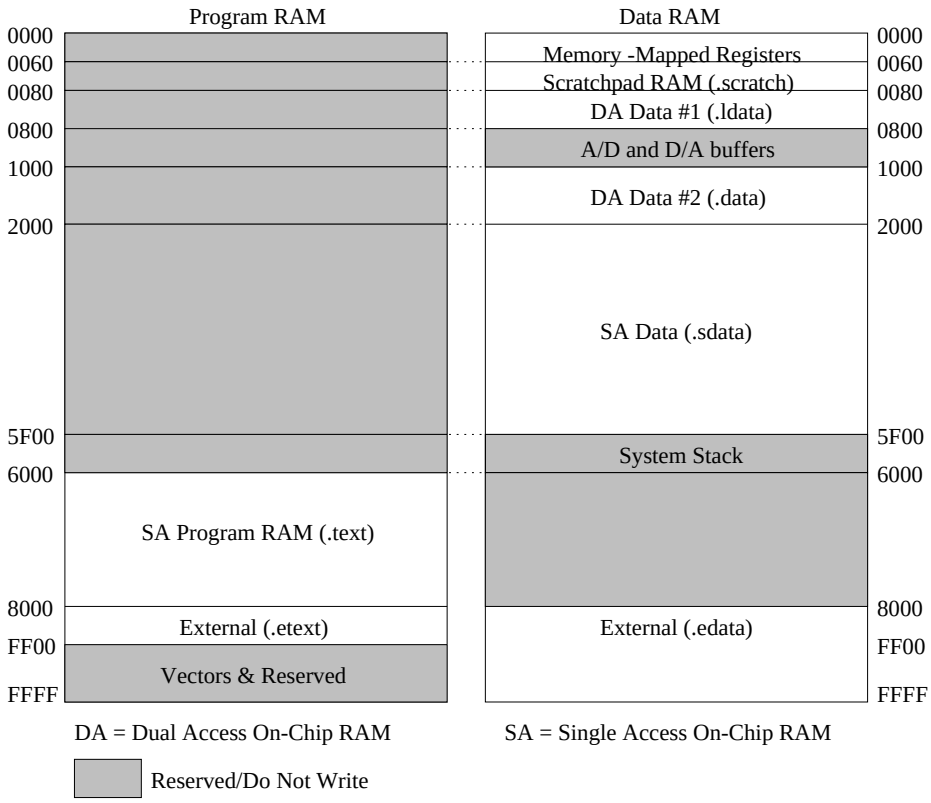

Figure 3: Linker Memory Map and Section Names

5 Using the Core File

5.1 Using the Audio D/A and A/D Converters

The following sample code (available as `thru6a.asm` in the `V:\ece320\54x\dsplib` directory) is the minimal amount of code required to send both of the input channels to three of the output channels on the six-channel board.

```
1           .copy "v:\ece320\54x\dsplib\core.asm"
2
3           .sect ".text"
4   main
5       ; Your initialization goes here.
6
7   loop
8       ; Wait for a new block of 64 samples to come in
```

```
 9       WAITDATA
10
11       ; BlockLen = the number of samples that come from WAITDATA (64)
12       stm #BlockLen-1, BRC        ; Repeat BlockLen=64 times
13       rptb block-1                ; ...from here to the ``block'' label
14
15       ld      *AR6,16, A          ; Receive ch1
16       mar *+AR6(2)                ; Rcv data is in every other word
17       ld      *AR6,16, B          ; Receive ch2
18       mar *+AR6(2)                ; Rcv data is in every other word
19
20       ; Code to process samples goes here.
21
22       sth a, *AR7+                ; Store into output buffer, ch1
23       sth a, *AR7+                ; ch2
24       sth a, *AR7+                ; ch3
25
26       sth b, *AR7+                ; Store into output buffer, ch4
27       sth b, *AR7+                ; ch5
28       sth b, *AR7+                ; ch6
29
30   block
31       b loop
```

Line 1 copies in the core code, which initializes the 6-channel board and the serial interface, provides the interface macros, and then jumps to "main" in your code. Line 3 declares that what follows should be placed in the program-code area in internal memory.

On Line 4, we find the label "main". This is the entry point for your code; once the DSP has finished initializing the 6-channel board and the serial port, the core file jumps to this label.

On Line 9, there is a call to `WAITDATA`. `WAITDATA` waits for the next block of 64 samples to arrive from the A/D. When it returns, a pointer to the samples captured by the A/D is returned in `AR6` (which can also be referred to as `pINBUF`); a pointer the start of the output buffer is returned in `AR7` (also `pOUTBUF`). Note that `WAITDATA` simply calls the "wait_fill" subroutine in the core file, which uses the `B` register internally, along with the `DP` register and the `TC` flag; therefore, do not expect the value of `B` to be preserved across the `WAITDATA` call.

Lines 12 and 13 set up a block repeat. BlockLen is set by the core code as the length of a block; the repeat instruction therefore repeats for every sample time. Lines 15-18 retrieve one sample from each of the two channels; note that the received data is placed in every other memory location. Lines 22-24 place the first input channel into the first three output channels, and lines 26-28 place the second input channel into the last three output channels. Figure 1 shows the relationship between the channel numbers shown in the code and the inputs and outputs on the surround-sound board.

Line 31 branches back to the top, where we wait for the next block to arrive and start over.

5.2 Using Test Vectors

A second version of the core file offers the same interface as the standard core file, but instead of reading input samples from the surround-sound board's A/D converters and sending output samples to the D/A converters, it reads and writes from test vectors generated in MATLAB.

Test vectors are a method for testing your code with known input. Given this known input and the specifications of the system, we can use simulations to determine what the output should be. We can then compare the expected output with the system's actual output. If the system is functioning properly, the expected output and actual output should match. (Will the expected output and the actual output from the DSP system match perfectly? Why or why not?)

Testing your system with test vectors may seem silly, because you can see if these simple filters work by looking at the output on the oscilloscope as you change the input frequency. However, they will become more useful as you write more complicated code. With more complicated DSP algorithms, testing becomes more difficult; when you correct an error that results in one case not working, you may introduce an error that causes another case to work improperly. This may not be immediately visible if you simply look at the oscilloscope and function generator; the oscilloscope doesn't display the signal continuously and transient errors may be hidden. In addition, it is easy to forget to check all possible input frequencies by sweeping the function generator after making a change.

More importantly, the test vectors also allow you to test signals that can't be generated or displayed with the oscilloscope and function generator. One important signal that can't be generated or tested with the function generator and oscilloscope is the impulse function; there is no way to view the impulse response of a filter directly without using test vectors. The unit impulse represents a particularly good test vector because it is easy to compare the actual impulse response of a digital filter against the expected impulse response. Testing using the impulse response also exposes the entire range of digital frequencies, unlike testing using periodic waveforms generated by the function generator.

Last, testing using test vectors allows us to isolate the digital signal processor from the analog input and output section. This is useful because the analog sections have some limitations, including imperfect anti-aliasing and anti-imaging filters. Testing using test vectors allows us to ensure that what we see is due only to the digital signal processing system, and not imperfections in the analog signal or electronics.

After generating a test vector in MATLAB, save it to a file that can be brought into your code using the MATLAB command `save_test_vector`:

```
>> save_test_vector('testvect.asm',ch1_in,ch2_in);  % Save test vector
```

(`ch2_in` can be omitted, in which case both channels of the test-vector input will have the same data.)

Next, modify your code to include the test-vector file you've created and the test-vector support code. This can be done by replacing the first line of the file — where the "core.asm" file is copied in — with two lines. Instead of:

```
1           .copy   "v:\ece320\54x\dsplib\core.asm"
```

use:

```
1           .copy   "testvect.asm"
2           .copy   "v:\ece320\54x\dsplib\vectcore.asm"
```

Note that, as usual, the whitespace in front of the `.copy` directive is required.

The test vector in the ".etext" section of program memory between 08000h and 0FEFFh. If you do not use this section, it will not interfere with your program code or data. This memory block is large enough to hold a test vector of up to 4,000 elements. Both channels of input, and all six channels of output, are stored in each test vector element.

These changes the the DSP assembly code copy in the test vector you created, and use an alternate core file to run the filter on the test vector. Next, assemble and load the file and reset and run as usual. After a few seconds, halt the DSP (using the Halt command under the Debug window) and verify that the DSP has halted at a branch statement that branches to itself: `spin b spin`.

Next, the test should be saved and loaded back into MATLAB. This is done by saving $6 * k$ memory elements (where k is the length of the test vector in samples) starting with location 0x8000 in program memory. Do this by choosing `File->Data->Save...` in Code Composer, then entering the filename `output.dat` and pressing `Enter`. Next, enter `0x8000` in the Address field of the dialog box that pops up, $6 * k$ in the Length field, and choosing "Program" from the drop-down menu next to "Page." (Always ensure that you use the correct length — 6 times the length of the test vector — when you save your results.)

Last, use the `read_vector` function to read the saved test vector output into MATLAB. Do this using the following MATLAB command:

```
>> [ch1, ch2, ch3, ch4, ch5, ch6] = read_vector('output.dat');
```

The MATLAB vectors `ch1` through `ch6` now contain the output of your program code in response to the input from the test vector.

5.3 Using External Program Memory

Because the 6-channel core file switches the on-chip memory into Program RAM to improve performance, the memory regions from 00000h to 007FFFh, 010000h to 017FFFh, and 020000h to 027FFFh in external RAM are not accessible to your code. This is because the on-chip memory is mapped into all of these locations when it is mapped into program space.

For this reason, special macros are provided by the 6-channel core file to access external program RAM. These macros call routines located in external RAM (in the reserved region between 0FF00h and 0FFFFh) that remove the on-chip RAM from the program memory space, copy one or more words of data between Program RAM and Data RAM, then re-map the on-chip RAM and return.

These two macros are READPROG and WRITPROG. READPROG copies words from external program RAM to data memory; WRITPROG copies words from data memory to external program RAM. Both macros take one argument (the number of words to copy; for instance, calling READPROG 1 copies one word from program memory to data memory). Before READPROG or WRITPROG is called, the starting address of the words in program RAM must be stored in the A register; the starting address of the words in data space is stored in address register AR1. After the call to READPROG or WRITPROG, A is unchanged, but AR1 points to the location after the last one written.

The following sample code demonstrates the use of READPROG and WRITPROG.

```
1   ; The following code copies 064h and 065h in Data RAM to 023456h and 023457h
2   ; in external Program RAM, then copies the data back to 074h and 075h in Data RAM.
3
4           stm     #64h,AR1        ; load 64h into AR1
5           ld      #02h,16,A       ; load 02h in high part of A
6           adds    #3456h,A        ; fill in low part of A
7                                   ; A contains 00023456h
8           WRITPROG 2              ; write from 064h and 065h into
9                                   ; 023456h and 023457h in ext Program RAM
10
11  ; Here, AR1 points to 066h and A still contains 00023456h
12
13          stm     #74h,AR1        ; load 74h into AR1
14
15          READPROG 2              ; read from 023456h and 023457h in
16                                  ; ext Program RAM into 074h and 075h
17
18  ; Here, AR1 points to 076h.
19
20
```

Do not use the external program memory region between 00FF00h and 00FFFFh for data storage. This area is used by the DSP to hold interrupt vectors and by the core file to hold the routines that copy data between the external RAM and data memory. If you overwrite this area, your code will not run properly.

5.4 Using the Serial Port

The 6-channel core file also supports the serial port installed on the DSP320C549 DSP. The serial port on the EVM is attached to COM2 on the PC. Before jumping to your code, the core file initializes the EVM's serial port to 38,400 bits per second with no parity, 1 stop bit and 8 data bits. It then accepts characters received from the PC by the UART (Universal Asynchronous Receiver/Transmitter) and buffers them in memory until your code retrieves them. It also can accept a block of bytes to transmit and send them to the UART in sequence.

Two macros are used to control the serial port: READSER and WRITSER. Both accept one parameter. READSER <n> reads up to <n> characters from the serial input buffer (the data coming from the PC), and places them in memory starting at *AR3. (AR3 is left pointing one past the last memory location written.) The actual number of characters read is left in AR1. If AR1 is zero, then no characters were available in the input buffer.

WRITSER <n> adds <n> characters starting at *AR3 to the serial output buffer - in other words, it queues them

to be sent to the PC. AR3 is left pointing one location after the last memory location read.

Note that READSER and WRITSER modify registers AR0, AR1, AR2, AR3, and BK as well as the flag TC. Be sure you restore these registers after calling READSER and WRITSER if you need them later in your code.

Note also that the 6-channel core file allows up to 126 characters to be stored in the input and output buffers. No checks to protect against buffer overflows are made, so do not allow the input and output buffers to overflow. (The length of the buffers can be changed by changing ser_rxlen and ser_txlen values in the core.asm file.) The buffers are 127 characters long; however, the code cannot distinguish between a completely-full and completely-empty buffer. Therefore, only 126 characters can be stored in the buffers.

It is easy to check if the input or output buffers in memory are empty. The input buffer can be checked by comparing the values stored in the memory locations srx_head and srx_tail; if both memory locations hold the same value, the input buffer is empty. Likewise, the output buffer can be checked by comparing the values stored in memory locations stx_head and stx_tail. The number of characters in the buffer can be computed by subtracting the head pointer from the tail pointer; add the length of the buffer (normally 127) if the resulting distance is negative.

The following example shows the minimal amount of code necessary to echo received data back through the serial port. It has been placed in V:\ece320\54x\dsplib\ as ser_echo.asm.

```
1           .copy "v:\ece320\54x\dsplib\core.asm"
2
3           .sect ".data"
4   hold    .word 0
5
6           .sect ".text"
7   main
8           stm #hold,AR3           ; Read to hold location
9
10          READSER 1               ; Read one byte from serial port
11
12          cmpm AR1,#1             ; Did we get a character?
13          bc main,NTC             ; if not, branch back to start
14
15          stm #hold,AR3           ; Write from hold location
16          WRITSER 1               ; ... one byte
17
18          b main
```

On Line 8, we tell READSER to receive into the location hold by setting AR3 to point at it. On Line 9, we call READSER 1 to read one serial byte into hold; the byte is placed in the low-order bits of the word and the high-order bits are zeroed. If a byte was read, AR1 will be set to 1. AR1 is checked in Line 12; Line 13 branches back to the top if no byte was read. Otherwise, we tell reset AR3 to hold (since READSER moved it), then call WRITSER to send the word we received on Line 16. On Line 18, we branch back to the start to receive another character.

6 Using MATLAB to Control the DSP

One of MATLAB's features is its ability to quickly create a visual interface that lets you use standard GUI controls (such as sliders, checkboxes and radio buttons) to call MATLAB scripts. An example of such an interface is available in the `V:\ece320\MAT_INT` directory, which contains two files:

- `ser_set.m`: Initializes the serial port and user interface
- `wrt_slid.m`: Called when sliders are moved to send new data

6.1 Creating a MATLAB user interface

The following code (`ser_set.m`) initializes the serial port COM2, then creates a minimal user interface consisting of three sliders.

```
1   % ser_set: Initialize serial port and create three sliders
2
3   % Set serial port mode
4   !mode com2:38400,n,8,1
5
6   % open a blank figure for the slider
7   Fig = figure(1);
8
9   % open sliders
10
11  %  first slider
12  sld1 = uicontrol(Fig,'units','normal','pos',[.2,.7,.5,.05],...
13   'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
14
15  %  second slider
16  sld2 = uicontrol(Fig,'units','normal','pos',[.2,.5,.5,.05],...
17   'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
18
19  %  third slider
20  sld3 = uicontrol(Fig,'units','normal','pos',[.2,.3,.5,.05],...
21   'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
22
23
```

Line 4 of this code uses the Windows NT `mode` command to set up COM port 2 (which is connected to the DSP) to match the serial port settings on the DSP evaluation board: 38,400 bps, no parity, 8 data bits and 1 stop bit. Line 7 then creates a new MATLAB figure for the controls; this prevents the controls from being overlaid on to of any graph you may have already created.

Lines 12 through the end create the three sliders for the user interface. Several parameters are used to specify the behavior of each slider. The first parameter, `Fig`, tells the slider to create itself in the window we created in Line 7. The rest of the parameters are property/value pairs:

- `units`: `Normal` tells MATLAB to use positioning relative to the window boundaries.
- `pos`: Tells MATLAB where to place the control.
- `style`: Tells MATLAB what type of control to place. `slider` creates a slider control.
- `value`: Tells MATLAB the default value for the control.
- `max`: Tells MATLAB the maximum value for the control.
- `min`: Tells MATLAB the minimum value for the control.
- `callback`: Tells MATLAB what script to call when the control is manipulated. `wrt_slid` is a MATLAB file that writes the values of the controls to the serial port.

6.2 The User Interface Callback Function

Every time a slider is moved, the `wrt_slid.m` file is called:

```
1   % wrt_slid: write values of sliders out to com port
2
3   % open com port for data transfer
4   fid = fopen('com2:','w');
5
6   % send data from each slider
7   v = round(get(sld1,'value'));
8   fwrite(fid,v,'int8');
9
10  v = round(get(sld2,'value'));
11  fwrite(fid,v,'int8');
12
13  v = round(get(sld3,'value'));
14  fwrite(fid,v,'int8');
15
16  % send reset pulse
17  fwrite(fid,255,'int8');
18
19  % close com port connection
20  fclose(fid);
```

Line 4 of `wrt_slid.m` opens COM2 for writing. (It has already been initialized.) Then Line 7 retrieves the value from the slider using MATLAB's `get` function to retrieve the `value` property. The value is then rounded off to create an integer, and the integer is sent as an 8-bit quantity to the DSP in Line 8. (The number that is sent at this step will appear when the serial port is read with `READ_SER` in your code.) Then the other two sliders are sent in the same way.

Line 17 sends 0xFF (255) to the DSP, which can be used to indicate that the three previously-transmitted values represent a complete set of data points. This can be used to prevent the DSP and MATLAB from losing synchronization if a transmitted character is not received by the DSP.

Line 20 closes the serial port. Note that MATLAB buffers the data being transmitted, and data is often not sent until the serial port is closed. Make sure you close the port after sending a data block to the DSP.

7 Bringing It Together: thru6.asm

The following code shows how to simultaneously use the A/D and D/A converters and the serial port. It is the "thru6.asm" code which performs both the A/D to D/A copy and the serial-port echo.

```
1           .copy "v:\ece320\54x\dsp_lib\core.asm"
2
3           .sect ".data"
4   hold    .word 0
5
6           .sect ".text"
7   main
8
9      ; Initialization code goes here.
10
11  loop
12      ; Wait for a new block of 64 samples to come in
13      WAITDATA
14
15      ; BlockLen = the number of samples that come from WAITDATA (64)
16      stm #BlockLen-1, BRC        ; Repeat BlockLen=64 times
17      rptb block-1                ; ...from here to the ``block'' label
18
19      ld      *AR6,16, A          ; Receive ch1
20      mar *+AR6(2)                ; Rcv data is in every other word
21      ld      *AR6,16, B          ; Receive ch2
22      mar *+AR6(2)                ; Rcv data is in every other word
23
24      ; Code to process samples goes here.
25
26      sth a, *AR7+                ; Store into output buffer, ch1
27      sth a, *AR7+                ; ch2
28      sth a, *AR7+                ; ch3
29
30      sth b, *AR7+                ; Store into output buffer, ch4
31      sth b, *AR7+                ; ch5
32      sth b, *AR7+                ; ch6
33
34  block
35          ; here we attempt to do serial stuff
36
37  _serial_loop
38          stm #hold,AR3           ; Read to hold location
39
40          READSER 1               ; Read one byte from serial port
41
42          cmpm AR1,#1             ; Did we get a character?
43          bc main,NTC             ; if not, branch back to start
44
45          stm #hold,AR3           ; Write from hold location
46          WRITSER 1               ; ... one byte
47
48          b _serial_loop
```